LAB 6 DAA

Auteurs

- Duruz Florian
- Ferreira Silva Sven
- Richard Aurélien

Groupe : DAA_B_12

Classe : DAA 2025-26

Date : 11/01/2026

Implémentation

Tout d'abord, nous avons décidé de ne pas utiliser les layout mais d'utiliser Jetpack compose pour l'application. Ce rapport va donc parler de l'implémentation demandée en utilisant Jetpack compose.

Utilisation du service REST

Afin d'utiliser le service REST mis à disposition, le fichier `ContactApiService.kt` fut créé. Il contient toute l'interaction avec l'endpoint `https://daa.iict.ch/`. Pour interagir avec le serveur, Ktor fut utilisé et ajouté au projet. Les appels aux endpoint se font de manière très simple :

```
    private val client = HttpClient(Android) {
        install(ContentNegotiation) {
            json(Json {
                ignoreUnknownKeys = true
                prettyPrint = true
                isLenient = true
            })
        }
        engine {
            connectTimeout = 5_000
            socketTimeout = 5_000
        }
    }
    private val baseUrl = "https://daa.iict.ch"

    suspend fun getContacts(uuid: String): List<ContactDTO> {
        return client.get("$baseUrl/contacts") {
            header("X-UUID", uuid)
        }.body()
    }
```

Pour effectuer certains de ces appels il faut fournir un UUID. Ce dernier est fourni par le service REST lors de l'appel à `enroll` et est stocké dans les `sharedPreferences` pour être persisté.

Entity Contact & DTO

Afin de gérer la syncrhonisation avec le serveur (ajout, suppression, modification) un nouvel enum `SyncState` fut ajouté à l'entité `Contact`. Il contient les états *dirty* (`CREATED`, `UPDATED`, `DELETED`) ainsi que l'état *pur* `SYNCED`. Ces états sont utilisé pour la synchronisation décalée.

Le champ `remoteId` fut aussi ajouté pour stocker l'id du contact sur le serveur.

Il n'est pas possible d'envoyer l'entity serialisée au serveur. Pour se faire la classe `ContactDTO` fut créée ainsi que des convertisseurs pour passer d'`Entity` à `DTO` et inversement. La classe `ContactDTO` contient principalement les mêmes informations que l'entité mais diffère sur le format de la date d'anniversaire et de son id (ici son id est le remoteId). Le `SyncState` n'est pas envoyé non plus (inutile pour le serveur)

ContactsDao

Le fichier `ContactsDao` fut changé en modifiant et ajoutant de nouvelles fonctions.
Les focntions `getContacts` et `getContactsAsList` furent ajoutés afin de pouvoir filtrer sur le `SyncState` des contacts. Par défaut seul l'état `DELETED` n'est pas séléctionné (on ne souhaite pas retourner un contact considéré comme supprimé). Il était aussi possible de filtrer en excluant `DELETED` (eg : 'SyncState != DELETED) -> blacklist. Mais le choix fut fait de filtrer avec une whitelist par défaut. Cela permet par la suite de pouvoir filtrer en passant en arguments les états souhaités.

La fonction `getCount` a eu le droit au même filtre !

ContactsRespository

Le repository prend plus de paramêtres ! Il prend en plus le service api défini plus haut ainsi qu'un `CoroutineDispatcher` (par défaut `Dispatcher.IO` vu les opérations effectuées).

La variable `allContacts` récupere maintenant ses données de `getContacts`.

Ensuite les fonctions `createContact`, `updateContact`, `deleteContact` vont effectuer leurs actions respectivent puis ensuite tenter de mettre à jour le serveur via le servic api. Si un succès est reçu alors on met à jour l'enregistrement dans room (par exemple passer le status `UPDATED` à `SYNCED` lors d'une mise à jour d'un contact).

La fonction `synchronize` va récuperer une liste de contacts ayant des status *dirty* (UPDATED, CREATED, DELETED) et va tenter de mettre à jour le serveur comme pour les fonctions d'avant.

A noter l'utilisation de `contactsDao.getContactsAsList` dans cette fonction ! Comme la logique dans le Dao était une whitelist, il est ici possible de passer les états souhaités et obtenir la liste de contact contenant uniquement ces états (dans notre cas, les états *dirty* (donc DELETED compris et SYNCED non))

Les fonctions `enroll` et `fetchContacts` vont simplement appeler le service et retourner/enregistrer les contacts. Cela permet au repository d'être la single source of truth. Le viewmodel ne va par exemple jamais appeler l'api, c'est le repo qui va le faire.

ContactsViewModel

Le viewmodel va utiliser les shared preferences de l'application afin de stocker le `token` d'enrollment retourné par le serveur lors de l'appel à `enroll`. Cela va permettre de le persister et donc de pouvoir survivre à la fermeture de l'application.

Afin de pouvoir "switch" entre les écrans de compose dans notre `MainActivity` (switch entre la liste de contact ou l'édition de contact) nous utilisons une variable `MutableStateFlow _editionMode` :

```
    private val _editionMode = MutableStateFlow(false)
    val editionMode: StateFlow<Boolean> = _editionMode.asStateFlow()
```

Cette dernière va permettre à l'écran de pouvoir gérer ses vues (on en reparlera plus tard).

Une variable du même style `_selectedContact` fut ajoutée, plus d'informations plus tard aussi.

Ces `MSF` vont pouvoir gérer l'état dans lequel l'application se trouve.
Grâce aux fonctions `openEditor` et `closeEditor` par exemple. L'écran implémentant le ViewModel va pouvoir appeler ces fonctions pour changer son 'état' et lorsque l'état change alors la vue change aussi car l'écran observe ces variables et va se recomposer en tenant compte des valeurs de ces dernières.

Le reste des fonctions sont les fonctions permettant d'interagir avec le repository (save, delete, enroll, etc...) en passant en plus le `token` lorsqu'il y'en a besoin. Ces fonctions sont lancées dans le scope du viewmodel.

ScreenContactEditor

L'écran de modification/création de contacts.

Afin de respecter le state hoisting, cet écran ne prend pas de ViewModel en paramêtres mais des callback (ainsi qu'un potentiel contact)! ces callbacks seront appelés lors de la fin d'édition du contact (`onCancel`. `onSave`, `onDelete`) via des boutons.

Toutes les informations du contact sont stockés localement dans des vars, c'est l'état interne de l'écran. Une fonction pour créer un `Contact` fut aussi ajoutée pour aléger le code.

La grand partie du code est l'UI. La majorité des champs sont des `TextField` et `Button`. Seulement le `DateField` a une fonction à part pour isoler la logique de création.

Si un contact est fourni, alors l'interface sera automatiquement remplie et le bouton `DELETE` apparaitra et le bouton pour sauvegarder se nommera `Save`.

Si à l'inverse aucun contact n'est fourni, les champs seront vide et le bouton DELETE n'apparaitra pas. Le bouton pour sauvgarder se nommera `Create`.

Des fonctions de `@Preview` sont présentes, elles furent surtout utilisées pour le développement.

AppContacts

L'écran principal, celui qui va utiliser le ViewModel.

Il utilise les variables `editionMode` et `selectedContact` afin de gérer ses vues. Elles sont récupérées du ViewModel grâce à `collectAsStateWithLifecycle()`. `editionMode` sert à basculer entre

`ScreenContactList` et `ScreenContactEditor` alors que `selectedContact` sert à conserver un contact pour l'utiliser plus tard.

Par exemple, le callback de `ScreenContactList` :

```
ScreenContactList(contacts) { selectedContact ->
    contactsViewModel.openEditor(selectedContact)
}
```

Le contact séléctionné dans l'écran de liste sera transmis au viewmodel via la fonction `openEditor` :

```
fun openEditor(contact: Contact?) {
    _selectedContact.value = contact
    _editionMode.value = true
}
```

Cette fonction va passer l'application en mode édition mais aussi conserver le contact séléctionné.

Comme `AppContact` observe ces variables l'écran va recharger et afficher l'écran d'édition avec les infos du contact séléctionné.

```
if (editionMode){ // sera = true dans notre exemple
    ScreenContactEditor(contact = selectedContact, // le contact conservé
        onCancel = { contactsViewModel.closeEditor() }, // les callbacks passés
        onDelete = { selectedContact?.let { contactsViewModel.delete(it) } },
        onSave = { contact -> contactsViewModel.save(contact) })
}else{ ...
```

Pour respecter le state hoisting, des callbacks sont passés aux composants au lieu du ViewModel.